第2章 高速データ・インターフェースのための並列メモリ設計法

―A-DコンバータのLVDSとDDRデータ・キャプチャのためのFPGA内部メモリの設計

Minseok Kim

第1章ではFPGAを用いた高速データ・インターフェースの設計法について，リファレンス設計を用いて解説しました．本章では，A-DコンバータのLVDSインターフェースとDDRデータ・キャプチャのためのFPGA内部メモリの設計を中心に解説します．これを活用する事例として「オシロスコープとロジック・アナライザ」の機能設計を通して分かりやすく説明します．　（筆者）

1．FPGAを中心としたシステムの構成

今回設計するシステムは，低コストFPGA　Cyclone II EP2C8-256（米国Altera社）を搭載し，2チャネル高速A-DコンバータAT84AD001B（米国Atmel社）を使った2チャネル・オシロスコープ機能と，FPGAのディジタル入力ポートを使って16ビット（2 POD）ロジック・アナライザ機能を実現します．また，ARM7コアの1チップ・マイコンAT91SAM7XC（米国Atmel社）を用いることで，ユーザ・インターフェースと制御を行うようにします．

第1章で紹介したFPGAの高速データ・インターフェースを用いて取得したデータは，マイコンからSPIバスにより読み出され，マイコンはそのデータをUSBやEthernet経由でパソコンに転送することになります（回路図はCD-ROMに収録）．今回のメモリ設計における特徴は，以下のようになります．

①オシロスコープ性能：2チャネル，最大1Gサンプル/sのリアルタイム・サンプリング（リアルタイムに観測する実時間サンプリング）

②ロジック・アナライザ性能：16ビット（2 POD），最大800 Mサンプル/sの非同期サンプリング

③FPGA内部のメモリ構成：チャネル当たり4,096ワード（512ワードのデュアル・ポートRAMを八つ並列で構成）

④低コスト・ハードウェアの構成：低コストFPGA，1チップ・マイコンを使用

図1に，FPGAを中心とした周辺機器コントロールの仕組みを示します．FPGAは，A-Dコンバータからのアナログ・データとディジタル入力ポート（POD入力）からキャプチャされたディジタル・データのやり取りのほかにも，マイコンから周辺デバイスの制御をインターフェースする役割を果たします．

FPGAのメモリ設計における入力データと対応チャネルの関係は以下のようになります．

①アナログCH1：A-DコンバータのI-チャネルの8ビット→チャネル1

②アナログCH2：A-DコンバータのQ-チャネルの8ビット→チャネル2

③ディジタルPOD1：ディジタル入力の下位8ビット（8チャネル）→チャネル3

④ディジタルPOD2：ディジタル入力の上位8ビット（8チャネル）→チャネル4

上記のデータは，すべて8ビット・ワードになっているため，これから上記の順番で4チャネルとして扱うことになります．FPGAが内蔵するメモリの容量は165,888ビットであることから，チャネル当たり4,096サンプルが連続データとして保存できます．つまり，131,072（＝4,096×4

KeyWord　LVDS，DDR，EP2C8，AT84AD001B，AT91SAM7XC，リアルタイム・サンプリング，オシロスコープ，ロジック・アナライザ，リング・メモリ
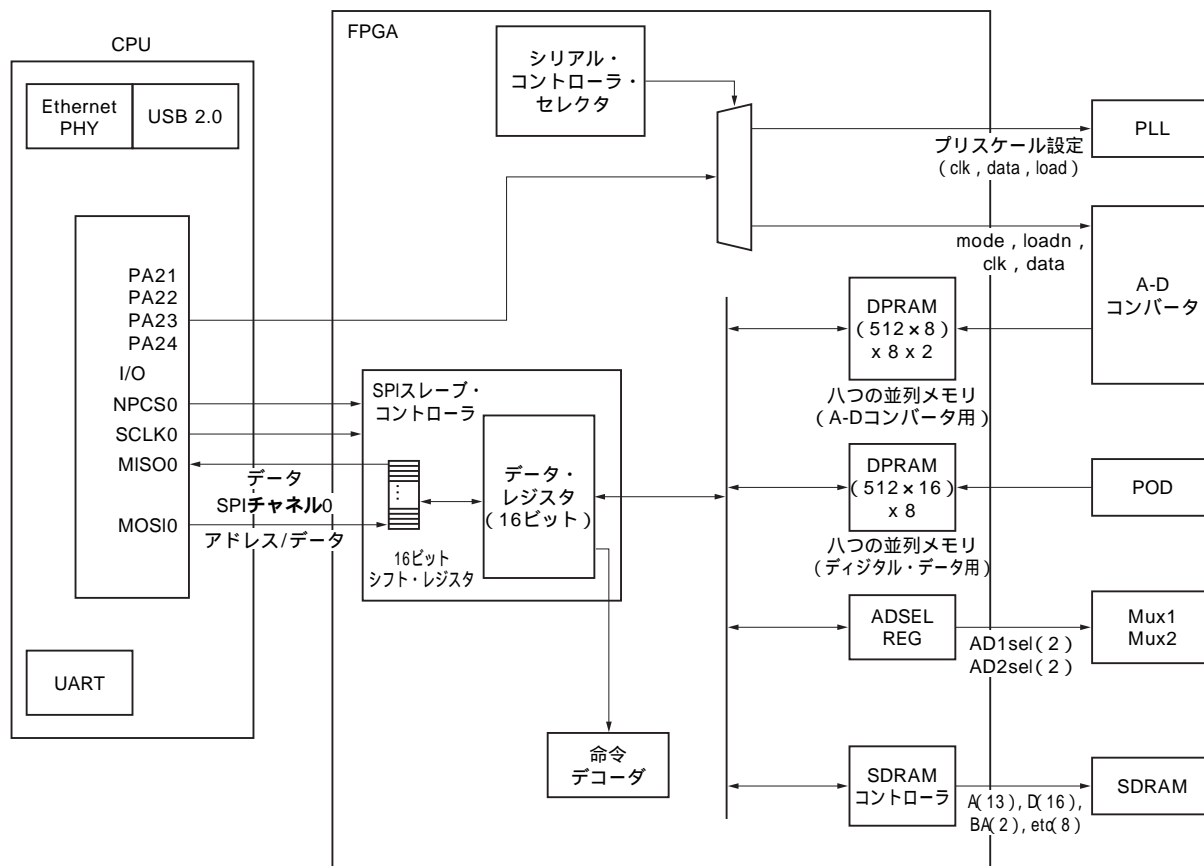
図1　FPGAを中心としたシステムのブロック・ダイアグラム
FPGAはA-Dコンバータからのアナログ・データとディジタル入力ポート（POD入力）からキャプチャされたディジタル・データのやり取りを行う．また，マイコンから周辺デバイスへのコントロールをインターフェースする．
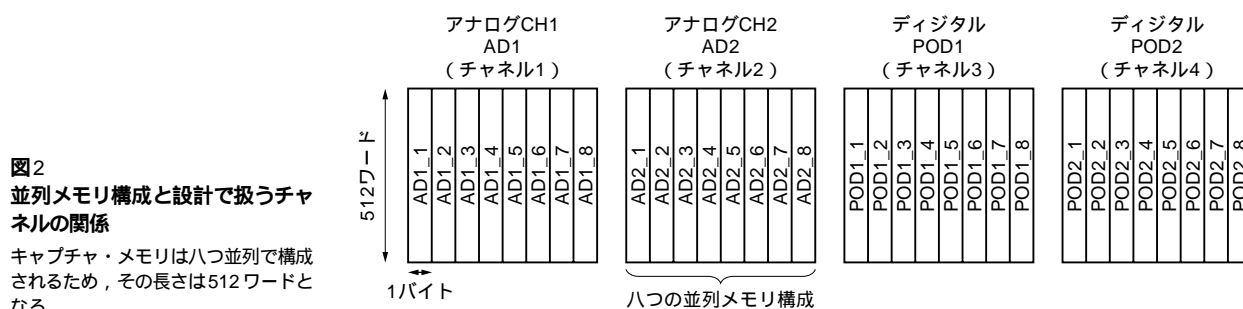
図2
並列メモリ構成と設計で扱うチャネルの関係
キャプチャ・メモリは八つ並列で構成されるため，その長さは512ワードとなる．

チャネル×8ビット）ビットを使用します．**図2**のようにキャプチャ・メモリは八つ並列で構成されるため，その長さは512ワードになります．

2．FPGAの内部メモリを用いた機能設計

第1章で紹介した方法を基礎に，高速データ・インターフェース設計に加えて，測定システムとして必要な機能を実装すれば，「オシロスコープ/ロジック・アナライザ」が実現できます．ここでは，実際にオシロスコープとロジック・アナライザの動作に必要な機能を簡単に設計することで，高速データ・インターフェースの理解を深めるとともに，測定システムの仕組みについても理解し，低コストFPGAでできることを体験してみます．
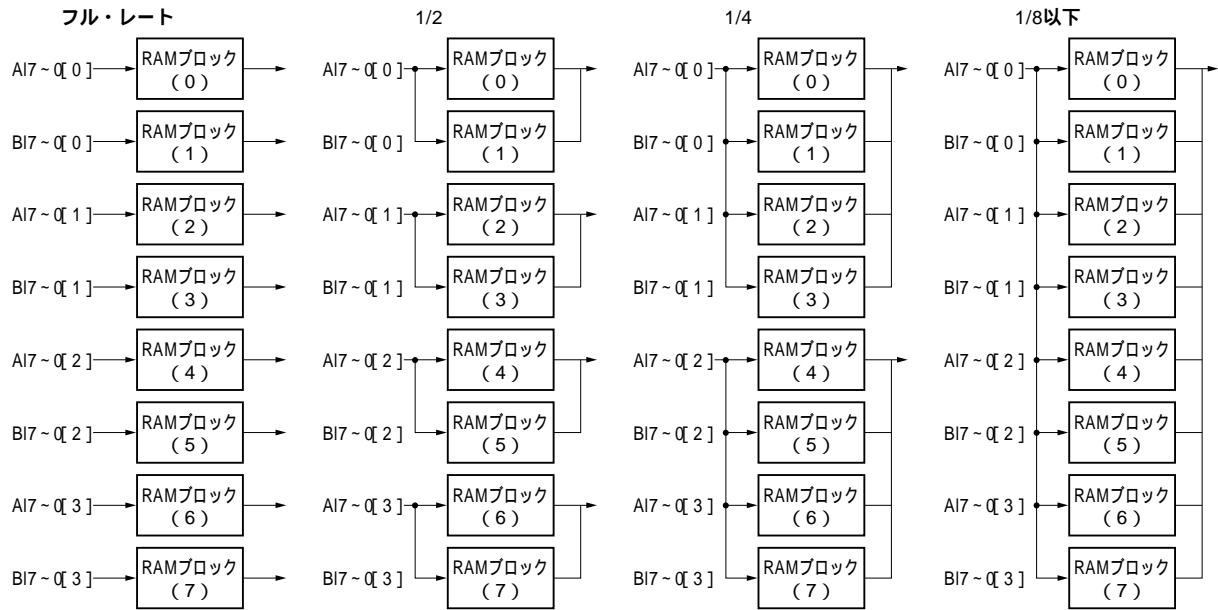
図3　データ書き込みの間引きによるレート可変化(チャネル1のみ)

サンプリング・レートが固定された場合，データのメモリ書き込みを必要とする周波数に合わせて間引く．八つのタイミングのデータを八つのメモリを用いて並列に書き込む構造であるため，メモリ実装が複雑になる．

● A-Dコンバータのメモリ書き込みレートを可変にする

オシロスコープ用として使用する高速A-Dコンバータは，高い周波数(今回は1GHz)のサンプリング・クロックを必要とします．これは，一般にVCO(voltage controlled oscillator；電圧制御発振器)とPLL(phase-locked loop；位相同期ループ)からなる外部のサンプリング・クロック生成回路より必要な周波数を作ります．

この場合，VCOの発振周波数はハードウェア的に固定してしまい，低周波数で長時間のデータを取得する場合は対応できなくなります．

オシロスコープとして使う場合，限られたメモリ資源をうまく使うために，書き込みレートの制御が必要になります．FPGAでサンプリング・クロックを管理できるなら，FPGAの内蔵PLLとクロック分周回路を使えば，問題はありません．高速A-Dコンバータの場合は，サンプリング・クロック周波数そのものを変えることはできないので，やや困ります．しかし，サンプリング・レート(物理的なVCO発振周波数)を固定のままにし，データのメモリ書き込みを必要とする周波数に合わせて間引くことで実現できそうです．

第1章の**図9**で示したように，八つのタイミングのデータを八つのメモリを用いて並列に書き込む構造になっているので，それを実現することは簡単ではなさそうです．

ここで，今回の設計条件に合わせた実装例を紹介します．**図3**にその動作イメージを示します．フル・レートの場合，それぞれのメモリに対して並列に書き込みを行います．書き込みレートを1/2，1/4，1/8以下に変更したい場合は，メモリ書き込み回路のデータ・パスの切り替え選択によって書き込みレートを制御し，むだなくメモリを利用することができます．

図4にいちばん簡単な例であるハーフ・レート(1/2)の場合の回路構成を示します．実装する回路はフル，1/2，1/4，1/8以下の4通りであり，1/8以下の場合は時間的に間引いて書き込みを行うことにします．回路は主に以下のような構成になります．

①メモリ書き込み部
- ライト・アドレスによるライト・イネーブル制御
- ライト・データのマルチプレクサ

②メモリ読み出し部
- リード・アドレスによるリード・メモリ・セレクタ
- バースト・フレーム転送用のマルチプレクサ(CPUからの読み出しをバースト的に行う場合)

このような回路を実装することで，状況に応じて適切なサンプリング・レートの切り替えができるようになります．

リスト1に，このVHDL記述を示します．ここでは，フル・レートを“1GHz”とし，対応できる周波数は“500MHz”，
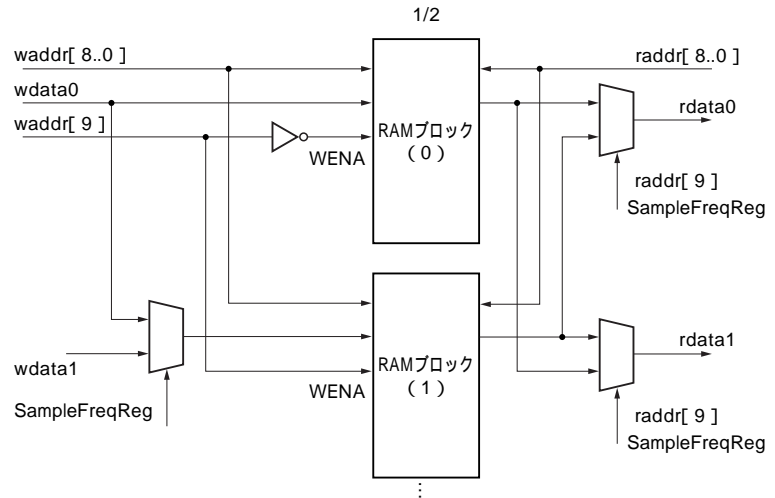
図4
A-D コンバータのメモリ書き込み/メモリ読み出し回路の多重化(1/2 モード)
すべてのメモリを100％使用しレートを可変にするには，書き込み部/読み出し部のデータ・パスの制御，アドレスの制御回路が必要．

リスト1　A-D コンバータ用の並列メモリ設計のVHDL記述(top.vhd モジュールから抜粋)
第1章のリスト1の回路をコンポーネントとして用いた並列メモリ操作回路．マルチプレクサの実装から書き込みレートを可変にすることができる(フル，1/2，1/4，1/8，1/8以下)．

```
(中略)
-- Iチャネル(CH1)用のLVDS SerDes （第1章，リスト 1)
    lvds_serdes_I_inst : lvds_serdes PORT MAP (
                inclock            => clkp(2),
                ADC_DIN            => ADCI_rx_data,
                Outclock           => ADCI_rx_outclock,
                locked             => ADCI_rx_locked,
                ADC_DOUT_A         => ADC_DIN_AI ,
                ADC_DOUT_B         => ADC_DIN_BI
      );

-- Qチャネル(CH2)用のLVDS SerDes （第1章，リスト 1)
    lvds_serdes_Q_inst : lvds_serdes PORT MAP (
                inclock            => clkp(2),
                ADC_DIN            => ADCQ_rx_data,
                Outclock           => ADCQ_rx_outclock,
                Locked             => ADCQ_rx_locked,
                ADC_DOUT_A         => ADC_DIN_AQ ,
                ADC_DOUT_B         => ADC_DIN_BQ
      );

--  Altera Megafunction デュアル・ポートRAMを用いた
                                          インスタント記述ループ
--  8ビット，512ワード
 dpram_loop3: FOR i IN 0 TO 3 GENERATE
   -- IチャネルAポート用DPRAM
    altera_ram_loop_A_I : altera_ram2p_8 PORT MAP (
                data       => ADC_DIN_AI_d(i+1),
                wren       => DPRAMCtrlADReg(1) AND
                                   ad_ram_I_wena(2*i+1),
                wraddress=> ad_wraddress_I(8 downto 0),
                rdaddress=> ad_rdaddress(8 downto 0),
                rden       => '1',
                wrclock    => ADCI_rx_outclock,
                rdclock    => rclk,
                rd_aclr    => DPRAMCtrlADReg(2),
                q          => dpram_out_ad_i(2*i+1)
      );

   -- IチャネルBポート用DPRAM
    altera_ram_loop_B_I : altera_ram2p_8 PORT MAP (
                data       => ADC_DIN_BI_d(i+1),
                wren       => DPRAMCtrlADReg(1) AND
                                   ad_ram_I_wena(2*i+2),
                wraddress=> ad_wraddress_I(8 downto 0),
                rdaddress=> ad_rdaddress(8 downto 0),
                rden       => '1',
                wrclock    => ADCI_rx_outclock,
                rdclock    => rclk,
                rd_aclr    => DPRAMCtrlADReg(2),
                q          => dpram_out_ad_i(2*i+2)
      );

   -- QチャネルAポート用DPRAM
    altera_ram_loop_A_Q : altera_ram2p_8 PORT MAP (
                data       => ADC_DIN_AQ_d(i+1),
                wren       => DPRAMCtrlADReg(1) AND
                                   ad_ram_Q_wena(2*i+1),
                wraddress=> ad_wraddress_Q(8 downto 0),
                rdaddress=> ad_rdaddress(8 downto 0),
                rden       => '1',
                wrclock    => ADCI_rx_outclock,
                rdclock    => rclk,
                rd_aclr    => DPRAMCtrlADReg(2),
                q          => dpram_out_ad_q(2*i+1)
      );

   -- QチャネルBポート用DPRAM
    altera_ram_loop_B_Q : altera_ram2p_8 PORT MAP (
                data       => ADC_DIN_BQ_d(i+1),
                wren       => DPRAMCtrlADReg(1) AND
                                   ad_ram_Q_wena(2*i+2),
                wraddress=> ad_wraddress_Q(8 downto 0),
                rdaddress=> ad_rdaddress(8 downto 0),
                rden       => '1',
                wrclock    => ADCI_rx_outclock,
                rdclock    => rclk,
                rd_aclr    => DPRAMCtrlADReg(2),
                q          => dpram_out_ad_q(2*i+2)
      );

    END GENERATE;

    -- 書き込みデータ・パスのマルチプレクサ回路
(サンプリング周波数設定により選択される)

    PROCESS (baseclock, SampleFreqReg) IS
      BEGIN
        CASE SampleFreqReg IS
          WHEN "0000" =>
            FOR i IN 0 to 3 LOOP
              ADC_DIN_AI_d(i+1) <= ADC_DIN_AI(i+1);
              ADC_DIN_BI_d(i+1) <= ADC_DIN_BI(i+1);

              ADC_DIN_AQ_d(i+1) <= ADC_DIN_AQ(i+1);
              ADC_DIN_BQ_d(i+1) <= ADC_DIN_BQ(i+1);
            END LOOP;

          WHEN "0001" =>
            FOR i IN 0 to 3 LOOP
              ADC_DIN_AI_d(i+1) <= ADC_DIN_AI(i+1);
              ADC_DIN_BI_d(i+1) <= ADC_DIN_AI(i+1);

              ADC_DIN_AQ_d(i+1) <= ADC_DIN_AQ(i+1);
```

リスト1 A-Dコンバータ用の並列メモリ設計のVHDL記述(top.vhdモジュールから抜粋)(つづき)

```
          ADC_DIN_BQ_d(i+1) <= ADC_DIN_AQ(i+1);
        END LOOP;

      WHEN "0010" =>
        FOR i IN 0 to 1 LOOP
          ADC_DIN_AI_d(i+1) <= ADC_DIN_AI(1);
          ADC_DIN_BI_d(i+1) <= ADC_DIN_AI(1);
          ADC_DIN_AI_d(i+3) <= ADC_DIN_AI(3);
          ADC_DIN_BI_d(i+3) <= ADC_DIN_AI(3);

          ADC_DIN_AQ_d(i+1) <= ADC_DIN_AQ(1);
          ADC_DIN_BQ_d(i+1) <= ADC_DIN_AQ(1);
          ADC_DIN_AQ_d(i+3) <= ADC_DIN_AQ(3);
          ADC_DIN_BQ_d(i+3) <= ADC_DIN_AQ(3);
        END LOOP;

      WHEN OTHERS =>
        FOR i IN 0 to 3 LOOP
          ADC_DIN_AI_d(i+1) <= ADC_DIN_AI(1);
          ADC_DIN_BI_d(i+1) <= ADC_DIN_AI(1);

          ADC_DIN_AQ_d(i+1) <= ADC_DIN_AQ(1);
          ADC_DIN_BQ_d(i+1) <= ADC_DIN_AQ(1);
        END LOOP;

    END CASE;
  END PROCESS;

-- 読み出しデータ・セレクト・マルチプレクサ
-- (サンプリング周波数設定とリード・アドレスにより選択される)

PROCESS (baseclock, SampleFreqReg) IS
  BEGIN
    CASE SampleFreqReg IS
      WHEN "0000" =>
          FOR j IN 0 to 7 LOOP
            dpram_out_ad(j+1) <= dpram_out_ad_q(j+1)
                               & dpram_out_ad_i(j+1);
          END LOOP;

      WHEN "0001" =>
      IF ad_rdaddress(9) = '1'THEN
          FOR j IN 0 to 3 LOOP
        dpram_out_ad(j+1) <= dpram_out_ad_q(2*j+2)
                              & dpram_out_ad_i(2*j+2);
        dpram_out_ad(j+5) <= dpram_out_ad_q(2*j+2)
                              & dpram_out_ad_i(2*j+2);
          END LOOP;

        ELSE
          FOR j IN 0 to 3 LOOP
        dpram_out_ad(j+1) <= dpram_out_ad_q(2*j+1)
                              & dpram_out_ad_i(2*j+1);
        dpram_out_ad(j+5) <= dpram_out_ad_q(2*j+1)
                              & dpram_out_ad_i(2*j+1);
          END LOOP;
        END IF;

      WHEN "0010" =>
          IF    ad_rdaddress(10 downto 9) = "00" THEN
          FOR j IN 0 to 1 LOOP
        dpram_out_ad(j+1) <= dpram_out_ad_q(4*j+1)
                              & dpram_out_ad_i(4*j+1);
        dpram_out_ad(j+3) <= dpram_out_ad_q(4*j+1)
                              & dpram_out_ad_i(4*j+1);
        dpram_out_ad(j+5) <= dpram_out_ad_q(4*j+1)
                              & dpram_out_ad_i(4*j+1);
        dpram_out_ad(j+7) <= dpram_out_ad_q(4*j+1)
                              & dpram_out_ad_i(4*j+1);
          END LOOP;

        ELSIF ad_rdaddress(10 downto 9) = "01" THEN
          FOR j IN 0 to 1 LOOP
        dpram_out_ad(j+1) <= dpram_out_ad_q(4*j+2)
                              & dpram_out_ad_i(4*j+2);
        dpram_out_ad(j+3) <= dpram_out_ad_q(4*j+2)
                              & dpram_out_ad_i(4*j+2);
        dpram_out_ad(j+5) <= dpram_out_ad_q(4*j+2)
                              & dpram_out_ad_i(4*j+2);
        dpram_out_ad(j+7) <= dpram_out_ad_q(4*j+2)
                              & dpram_out_ad_i(4*j+2);
          END LOOP;

        ELSIF ad_rdaddress(10 downto 9) = "10" THEN
          FOR j IN 0 to 1 LOOP
        dpram_out_ad(j+1) <= dpram_out_ad_q(4*j+3)
                              & dpram_out_ad_i(4*j+3);
        dpram_out_ad(j+3) <= dpram_out_ad_q(4*j+3)
                              & dpram_out_ad_i(4*j+3);
        dpram_out_ad(j+5) <= dpram_out_ad_q(4*j+3)
                              & dpram_out_ad_i(4*j+3);
        dpram_out_ad(j+7) <= dpram_out_ad_q(4*j+3)
                              & dpram_out_ad_i(4*j+3);
          END LOOP;

        ELSIF ad_rdaddress(10 downto 9) = "11" THEN
          FOR j IN 0 to 1 LOOP
        dpram_out_ad(j+1) <= dpram_out_ad_q(4*j+4)
                              & dpram_out_ad_i(4*j+4);
        dpram_out_ad(j+3) <= dpram_out_ad_q(4*j+4)
                              & dpram_out_ad_i(4*j+4);
        dpram_out_ad(j+5) <= dpram_out_ad_q(4*j+4)
                              & dpram_out_ad_i(4*j+4);
        dpram_out_ad(j+7) <= dpram_out_ad_q(4*j+4)
                              & dpram_out_ad_i(4*j+4);
          END LOOP;

        END IF;

      WHEN OTHERS =>
      IF      ad_rdaddress(11 downto 9) = "000" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(1) &
                               dpram_out_ad_i(1);
          END LOOP;

      ELSIF   ad_rdaddress(11 downto 9) = "001" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(2) &
                               dpram_out_ad_i(2);
          END LOOP;

      ELSIF   ad_rdaddress(11 downto 9) = "010" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(3) &
                               dpram_out_ad_i(3);
          END LOOP;

      ELSIF   ad_rdaddress(11 downto 9) = "011" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(4) &
                               dpram_out_ad_i(4);
          END LOOP;

      ELSIF   ad_rdaddress(11 downto 9) = "100" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(5) &
                               dpram_out_ad_i(5);
          END LOOP;

      ELSIF   ad_rdaddress(11 downto 9) = "101" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(6) &
                               dpram_out_ad_i(6);
          END LOOP;

      ELSIF   ad_rdaddress(11 downto 9) = "110" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(7) &
                               dpram_out_ad_i(7);
          END LOOP;

      ELSIF   ad_rdaddress(11 downto 9) = "111" THEN
          FOR j IN 1 to 8 LOOP
            dpram_out_ad(j) <= dpram_out_ad_q(8) &
                               dpram_out_ad_i(8);
          END LOOP;
        END IF;

    END CASE;
  END PROCESS;
```

リスト1　A-D コンバータ用の並列メモリ設計のVHDL 記述（top.vhd モジュールから抜粋）（つづき）

```
     -- 書き込みイネーブル用マルチプレクサ  （サンプリング周波数設定により選択される）
  -- チャネルI

      PROCESS (ADCI_rx_outclock, SampleFreqReg) IS
       BEGIN
       CASE SampleFreqReg IS
         WHEN "0000" =>
              FOR j IN 0 to 7 LOOP
                ad_ram_I_wena(j+1) <= '1';
              END LOOP;

         WHEN "0001" =>
              FOR j IN 0 to 3 LOOP
                ad_ram_I_wena(2*j+1) <= NOT ad_wraddress_I(9);
                ad_ram_I_wena(2*j+2) <=     ad_wraddress_I(9);
              END LOOP;

         WHEN "0010" =>
              FOR j IN 0 to 1 LOOP
                ad_ram_I_wena(4*j+1) <= bool2bit(ad_wraddress_I(10 downto 9) = "00");
                ad_ram_I_wena(4*j+2) <= bool2bit(ad_wraddress_I(10 downto 9) = "01");
                ad_ram_I_wena(4*j+3) <= bool2bit(ad_wraddress_I(10 downto 9) = "10");
                ad_ram_I_wena(4*j+4) <= bool2bit(ad_wraddress_I(10 downto 9) = "11");
              END LOOP;

         WHEN "0011" =>
                ad_ram_I_wena(1) <= bool2bit(ad_wraddress_I(11 downto 9) = "000");
                ad_ram_I_wena(2) <= bool2bit(ad_wraddress_I(11 downto 9) = "001");
                ad_ram_I_wena(3) <= bool2bit(ad_wraddress_I(11 downto 9) = "010");
                ad_ram_I_wena(4) <= bool2bit(ad_wraddress_I(11 downto 9) = "011");
                ad_ram_I_wena(5) <= bool2bit(ad_wraddress_I(11 downto 9) = "100");
                ad_ram_I_wena(6) <= bool2bit(ad_wraddress_I(11 downto 9) = "101");
                ad_ram_I_wena(7) <= bool2bit(ad_wraddress_I(11 downto 9) = "110");
                ad_ram_I_wena(8) <= bool2bit(ad_wraddress_I(11 downto 9) = "111");

         WHEN OTHERS =>
                ad_ram_I_wena(1) <= bool2bit(ad_wraddress_I(11 downto 9) = "000") AND deciflag_ADI;
                ad_ram_I_wena(2) <= bool2bit(ad_wraddress_I(11 downto 9) = "001") AND deciflag_ADI;
                ad_ram_I_wena(3) <= bool2bit(ad_wraddress_I(11 downto 9) = "010") AND deciflag_ADI;
                ad_ram_I_wena(4) <= bool2bit(ad_wraddress_I(11 downto 9) = "011") AND deciflag_ADI;
                ad_ram_I_wena(5) <= bool2bit(ad_wraddress_I(11 downto 9) = "100") AND deciflag_ADI;
                ad_ram_I_wena(6) <= bool2bit(ad_wraddress_I(11 downto 9) = "101") AND deciflag_ADI;
                ad_ram_I_wena(7) <= bool2bit(ad_wraddress_I(11 downto 9) = "110") AND deciflag_ADI;
                ad_ram_I_wena(8) <= bool2bit(ad_wraddress_I(11 downto 9) = "111") AND deciflag_ADI;
         END CASE;
      END PROCESS;

  -- 書き込みイネーブル用マルチプレクサ  （サンプリング周波数設定により選択される）
  -- チャネルQ

      PROCESS (ADCQ_rx_outclock, SampleFreqReg) IS
       BEGIN
       CASE SampleFreqReg IS
         WHEN "0000" =>
              FOR j IN 0 to 7 LOOP
                ad_ram_Q_wena(j+1) <= '1';
              END LOOP;

         WHEN "0001" =>
              FOR j IN 0 to 3 LOOP
                ad_ram_Q_wena(2*j+1) <= NOT ad_wraddress_Q(9);
                ad_ram_Q_wena(2*j+2) <=     ad_wraddress_Q(9);
              END LOOP;

         WHEN "0010" =>
              FOR j IN 0 to 1 LOOP
                ad_ram_Q_wena(4*j+1) <= bool2bit(ad_wraddress_Q(10 downto 9) = "00");
                ad_ram_Q_wena(4*j+2) <= bool2bit(ad_wraddress_Q(10 downto 9) = "01");
                ad_ram_Q_wena(4*j+3) <= bool2bit(ad_wraddress_Q(10 downto 9) = "10");
                ad_ram_Q_wena(4*j+4) <= bool2bit(ad_wraddress_Q(10 downto 9) = "11");
              END LOOP;
         WHEN "0011" =>
                ad_ram_Q_wena(1) <= bool2bit(ad_wraddress_Q(11 downto 9) = "000");
                ad_ram_Q_wena(2) <= bool2bit(ad_wraddress_Q(11 downto 9) = "001");
                ad_ram_Q_wena(3) <= bool2bit(ad_wraddress_Q(11 downto 9) = "010");
                ad_ram_Q_wena(4) <= bool2bit(ad_wraddress_Q(11 downto 9) = "011");
                ad_ram_Q_wena(5) <= bool2bit(ad_wraddress_Q(11 downto 9) = "100");
                ad_ram_Q_wena(6) <= bool2bit(ad_wraddress_Q(11 downto 9) = "101");
                ad_ram_Q_wena(7) <= bool2bit(ad_wraddress_Q(11 downto 9) = "110");
                ad_ram_Q_wena(8) <= bool2bit(ad_wraddress_Q(11 downto 9) = "111");
         WHEN OTHERS =>
                ad_ram_Q_wena(1) <= bool2bit(ad_wraddress_Q(11 downto 9) = "000") AND deciflag_ADQ;
```

リスト1 A-D コンバータ用の並列メモリ設計のVHDL記述(top.vhd モジュールから抜粋)(つづき)

```
                ad_ram_Q_wena(2) <=
bool2bit(ad_wraddress_Q(11 downto 9) = "001") AND
deciflag_ADQ;
                ad_ram_Q_wena(3) <=
bool2bit(ad_wraddress_Q(11 downto 9) = "010") AND
deciflag_ADQ;
                ad_ram_Q_wena(4) <=
bool2bit(ad_wraddress_Q(11 downto 9) = "011") AND
deciflag_ADQ;
                ad_ram_Q_wena(5) <=
bool2bit(ad_wraddress_Q(11 downto 9) = "100") AND
deciflag_ADQ;
                ad_ram_Q_wena(6) <= bool2bit(ad_wraddress_Q
                   (11 downto 9) = "101") AND deciflag_ADQ;
                ad_ram_Q_wena(7) <= bool2bit(ad_wraddress_Q
                   (11 downto 9) = "110") AND deciflag_ADQ;
                ad_ram_Q_wena(8) <= bool2bit(ad_wraddress_Q
                   (11 downto 9) = "111") AND deciflag_ADQ;
          END CASE;
      END PROCESS;

-- 周波数分周に使うマルチプレクサ
    freqdivider <=
-- 1/8以下の場合
     Conv_std_logic_vector(  1,11)  WHEN SampleFreqReg =
                                  "0100" ELSE  --  62.5M
     Conv_std_logic_vector(  3,11)  WHEN SampleFreqReg =
                                 "0101" ELSE  --  31.25M
     Conv_std_logic_vector(  4,11)  WHEN SampleFreqReg =
                                   "0110" ELSE  --  25M
     Conv_std_logic_vector(  9,11)  WHEN SampleFreqReg =
                                  "0111" ELSE  --  12.5M
     Conv_std_logic_vector( 19,11)  WHEN SampleFreqReg =
                                  "1000" ELSE  --  6.25M
     Conv_std_logic_vector( 24,11)  WHEN SampleFreqReg =
                                   "1001" ELSE  --  5M
     Conv_std_logic_vector( 49,11)  WHEN SampleFreqReg =
                                   "1010" ELSE  --  2.5M
     Conv_std_logic_vector( 99,11)  WHEN SampleFreqReg =
                                  "1011" ELSE  --  1.25M
     Conv_std_logic_vector(124,11)  WHEN SampleFreqReg =
                                   "1100" ELSE  --  1M
     Conv_std_logic_vector(249,11)  WHEN SampleFreqReg =
                                   "1101" ELSE  --  0.5M
     Conv_std_logic_vector(499,11)  WHEN SampleFreqReg =
                                  "1110" ELSE  --  0.25M
     Conv_std_logic_vector(999,11)  WHEN SampleFreqReg =
                                 "1111" ELSE  --  0.125M
     Conv_std_logic_vector(  0,11);

-- レート1/8以下の場合,飛び飛び書き込みを行うための
      ライト・イネーブル信号

    PROCESS (ADCI_rx_outclock) IS
      BEGIN
        IF ADCI_rx_outclock'event and ADCI_rx_outclock =
                                                  '0' THEN
          IF freqcount_ADI = freqdivider THEN
            freqcount_ADI <= (OTHERS=> '0');
            deciflag_ADI <= '1';
          ELSE
            freqcount_ADI <= freqcount_ADI + 1;
            deciflag_ADI <= '0';
          END IF;
        END IF;
      END PROCESS;

    PROCESS (ADCQ_rx_outclock) IS
      BEGIN
        IF ADCQ_rx_outclock'event and ADCQ_rx_outclock =
                                                  '0' THEN
          IF freqcount_ADQ = freqdivider THEN
            freqcount_ADQ <= (OTHERS=> '0');
            deciflag_ADQ <= '1';
          ELSE
            freqcount_ADQ <= freqcount_ADQ + 1;
            deciflag_ADQ <= '0';
          END IF;
        END IF;
      END PROCESS;

 -- サンプル数の設定(サンプリング周波数設定により選択される)

    PROCESS (baseclock, SampleFreqReg) IS
      BEGIN
       CASE SampleFreqReg IS
         WHEN "0000" =>  SampleNoReg <= "000011111111";
                                     -- フル・レートの場合に511
         WHEN "0001" =>  SampleNoReg <= "000111111111";
                                  -- ハーフ・レートの場合に1023
         WHEN "0010" =>  SampleNoReg <= "001111111111";
                                     -- 1/4レートの場合に2047
         WHEN OTHERS =>  SampleNoReg <= "011111111111";
                                -- 1/8以下のレートの場合に4095
              END CASE;
      END PROCESS;
(中略)
```

“250MHz”,“125MHz”,“62.5MHz”,“31.25MHz”,“25MHz”,“12.5MHz”,“6.25MHz”,“5MHz”,“2.5MHz”,“1.25MHz”,“1MHz”,“0.5MHz”,“0.25MHz”,“0.125MHz”になります.

●DDRディジタル・キャプチャのメモリ設計

A-Dコンバータのサンプリング・クロックはハードウェア的に固定されているため,データを間引くことでメモリ書き込みレートを制御しました.しかし,ベースになるクロック周波数を直接制御できる場合は,そのクロック周波数を変えるだけなので,回路が簡単で済みます.ロジック・アナライザ用に使うには,データのキャプチャ・レートを可変に設計する必要があります.

今回は,**図5**のようにキャプチャ・クロックの生成回路
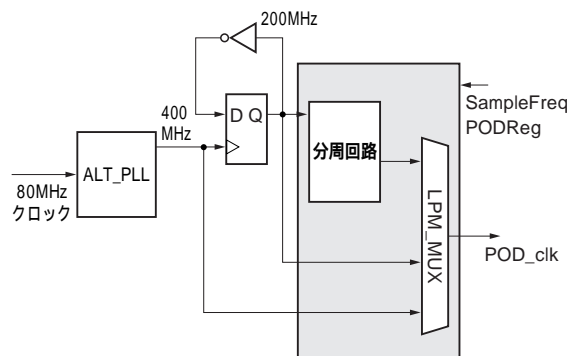
図5 クロック分周回路とグロバール・クロック・アサイン

PLLを用いて400MHzのフル・レート・クロックを生成する.分周回路からは1/4以下のクロックを生成することになり,フル・レートやハーフ・レートとともに選択可能なキャプチャ・クロック(POD_clk)として用意する.その選択回路でLPM_MUXを用いれば,出力のキャプチャ・クロック(POD_clk)はFPGA内部のグローバル・クロック・ネットワークに割り当てられる.

リスト2　クロック分周回路のVHDL記述（clockdiv.vhd）

カウンタを400MHzで動作させることは厳しいため，FF（Flip-flop）を用いて生成されたハーフ・レート・クロックを使う．分周回路から生成される1/4以下のクロックを，フル・レートやハーフ・レートとともに選択可能なキャプチャ・クロック（POD_clk）として用意する．その選択は米国Altera社提供のLPM_MUXを用いる．

```
--************************************--
--       clockdiv.vhd
--       by Minseok Kim
--************************************--

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_UNSIGNED.all;
USE work.util_package.ALL;

ENTITY clockdiv IS
  PORT (
    inclock       :IN std_logic;
    SampleFreqPOD :IN std_logic_vector(3 downto 0);

    outclock      :OUT std_logic;  -- for capture
    sysclock      :OUT std_logic;  -- for system (80MHz)
    locked        :OUT std_logic
);
END ENTITY clockdiv;

ARCHITECTURE rtl OF clockdiv IS

  COMPONENT altera_pll
      PORT
      (
              inclk0 : IN STD_LOGIC  := '0';
              pllena : IN STD_LOGIC  := '1';
              areset : IN STD_LOGIC  := '0';
              c0     : OUT STD_LOGIC ;
              c1     : OUT STD_LOGIC ;
              c2     : OUT STD_LOGIC ;
              locked : OUT STD_LOGIC
      );
  END COMPONENT;

  COMPONENT alt_mux
      PORT
      (
  --          clock  : IN STD_LOGIC ;
              data0  : IN STD_LOGIC ;
              data1  : IN STD_LOGIC ;
              data2  : IN STD_LOGIC ;
              data3  : IN STD_LOGIC ;
              sel    : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
              result : OUT STD_LOGIC
      );
  END COMPONENT;

  SIGNAL  pll_c0, pll_c1, pll_c2 : std_logic;
  SIGNAL  pll_c0_half                       : std_logic;
  SIGNAL  dbase_clk_POD                     : std_logic;
  SIGNAL  freqdividerPOD, freqcountPOD      :
                         std_logic_vector(10 downto 0) ;
  SIGNAL  SampleFreqPOD_sig              :
                         std_logic_vector( 1 downto 0) ;

BEGIN

    altera_pll_inst : altera_pll PORT MAP (
      inclk0             => inclock,
      pllena             => '1',
      areset             => '0',
      c0                 => pll_c0,
                  -- キャプチャ用ベース・クロック  (x4:400MHz)
      c1                 => pll_c1,
                 -- システム・クロック          (x4/5:80MHz)
      locked             => locked
    );


    freqdividerPOD <=                  -- SampleFreqPOD =
                                   "0000" ELSE  -- 400M
                                     -- SampleFreqPOD =
                                   "0001" ELSE  -- 200M
     Conv_std_logic_vector(   0,11)  WHEN SampleFreqPOD =
                                   "0010" ELSE  -- 100M
     Conv_std_logic_vector(   1,11)  WHEN SampleFreqPOD =
                                   "0011" ELSE  --  50M
     Conv_std_logic_vector(   3,11)  WHEN SampleFreqPOD =
                                   "0100" ELSE  --  25M
     Conv_std_logic_vector(   7,11)  WHEN SampleFreqPOD =
                                 "0101" ELSE  --  12.5M
     Conv_std_logic_vector(   9,11)  WHEN SampleFreqPOD =
                                   "0110" ELSE  --  10M
     Conv_std_logic_vector(  19,11)  WHEN SampleFreqPOD =
                                    "0111" ELSE  --  5M
     Conv_std_logic_vector(  39,11)  WHEN SampleFreqPOD =
                                  "1000" ELSE  --  2.5M
     Conv_std_logic_vector(  49,11)  WHEN SampleFreqPOD =
                                    "1001" ELSE  --  2M
     Conv_std_logic_vector(  99,11)  WHEN SampleFreqPOD =
                                    "1010" ELSE  --  1M
     Conv_std_logic_vector( 199,11)  WHEN SampleFreqPOD =
                                  "1011" ELSE  --  0.5M
     Conv_std_logic_vector( 249,11)  WHEN SampleFreqPOD =
                                  "1100" ELSE  --  0.4M
     Conv_std_logic_vector( 499,11)  WHEN SampleFreqPOD =
                                  "1101" ELSE  --  0.2M
     Conv_std_logic_vector( 999,11)  WHEN SampleFreqPOD =
                                  "1110" ELSE  --  0.1M
     Conv_std_logic_vector(1999,11)  WHEN SampleFreqPOD =
                                 "1111" ELSE  --  0.05M
     Conv_std_logic_vector(   0,11);  --  100M


    PROCESS (pll_c0) IS
      BEGIN
        IF pll_c0'event and pll_c0 = '1' THEN
          pll_c0_half <= NOT pll_c0_half;
        END IF;
      END PROCESS;

    PROCESS (pll_c0_half) IS
      BEGIN
        IF pll_c0_half'event and pll_c0_half = '1' THEN
          IF freqcountPOD >= freqdividerPOD THEN
            freqcountPOD <= (OTHERS=> '0');
            dbase_clk_POD <= NOT dbase_clk_POD;
          ELSE
            freqcountPOD <= freqcountPOD + 1;
          END IF;
        END IF;
      END PROCESS;

    PROCESS (SampleFreqPOD) IS
      BEGIN
        CASE SampleFreqPOD IS
            WHEN "0000" => SampleFreqPOD_sig <= "00";
            WHEN "0001" => SampleFreqPOD_sig <= "01";
            WHEN OTHERS => SampleFreqPOD_sig <= "10";
        END CASE;
      END PROCESS;

    -- base clock selector
    -- multiplexer output should be a global signal
    alt_mux_clock : alt_mux PORT MAP (
             data0  => pll_c0,           -- 400 MHz
             data1  => pll_c0_half,      -- 200 MHz
             data2  => dbase_clk_POD,    -- below 100 MHz
             data3  => dbase_clk_POD,
             sel    => SampleFreqPOD_sig,
             result => outclock);

    sysclock <= pll_c1;


END ;
```

を構成します．外部クロック（周波数は任意だが，ここでは80MHzのものを使用）からの入力クロックをFPGAの内部PLLに入力します．次に，周波数を400MHzにてい倍してベース・クロックとして使います．これは，Cyclone IIの内部PLLのほぼ最大出力周波数になります（FPGAの仕様上，最大周波数は402.5MHzである）．

クロック分周回路はカウンタで構成しますが，400MHzで動作することは厳しいため，フリップフロップを用いて生成されたハーフ・レート・クロックを使います．分周回路からは1/4以下のクロックを生成することになり，フル・レートやハーフ・レートとともに選択可能なキャプチャ・クロック（POD_clk）として用意します．その選択回路はマルチプレクサのマクロ（LPM_MUX，Altera社）を用いて実装します．LPM_MUXを用いれば，出力のキャプチャ・クロック（POD_clk）をFPGA内部のグローバル・クロック・ネットワークに割り当てることができます．

リスト2にディジタル・キャプチャのクロック生成（clockdiv.vhd）のVHDL記述を示します．“800MHz”をベース・クロック（400MHzクロックのダブル・レート）とし，対応の周波数は“400MHz”,“200MHz”,“100MHz”,“50MHz”,“25MHz”,“20MHz”,“10MHz”,“5MHz”,“4MHz”,“2MHz”,“1MHz”,“0.8MHz”,“0.4MHz”,“0.2MHz”,“0.1MHz”のようになります．

前章で説明したように，PODの論理値はDDRバッファによってクロック（POD_clk）の立ち上がりと立ち下がりの両エッジでキャプチャされ，それぞれのデータはクロックの立ち上がりエッジで同期出力されます．DDRバッファにより両エッジでキャプチャされたデータは1：4のDESER（deserializer）モジュールに入力され，最終的に八つの異なる位相でキャプチャしたデータが得られます．すべての出力データは，POD_qclkの立ち上がりエッジで同期します．

リスト3にディジタル・キャプチャのメモリ回路のVHDL記述を示します．

● トリガの実装とメモリ・アクセス

オシロスコープやロジック・アナライザのような計測システムにおいて，トリガは欠かせない機能です．トリガ（trigger；引き金という意味で何かを起動させること）とは，被測定信号のある条件が満たされたときに，そのタイミングの前後にある区間のデータを，メモリに取り込むことです．トリガ機能がなければ，測定器は被測定信号を適当なタイミングで指定された数量をキャプチャするだけになりますが，トリガを用いると変化に反応して意図した部分の測定を行うことができます．

ここで，オシロスコープとロジック・アナライザにおけるトリガの原理を簡単に解説し，FPGAを用いた簡易トリガの論理回路設計を行います．

リスト3　DDRディジタル・キャプチャのメモリ回路のVHDL記述（top.vhdから抜粋，第1章のリスト4の続き）

DDIOから両エッジでキャプチャされたデータは1：4のDESER（deserializer）モジュールに入力され，最終的に八つの異なる位相でキャプチャしたデータが得られる．すべての出力データは，POD_qclkの立ち上がりエッジで同期して八つのメモリに同時に書き込まれる．

```
（中略）

-- ディジタル・キャプチャのための，PLL/Clock回路記述
====================================

    clockdiv_inst : clockdiv PORT MAP (
      inclock            => clkp(1),
      SampleFreqPOD      => SampleFreqPOD_reg,
      Outclock           => POD_clk,
                      -- for sampling clock    (x4:400MHz)
      Sysclock           => baseclock,
                    -- for system baseclock  (x4/5:80MHz)
      Locked             => pll1_locked
    );

   -- 並列メモリ・ブロック
  dpram_loop1: FOR i IN 0 TO 3 GENERATE

    pod_ram1 : altera_ram2p PORT MAP (
               data       => POD_IN(2*i+1),
               wren       => DPRAM_POD_write_start,
               wraddress  => pod_wraddress,
               rdaddress  => pod_rdaddress,
               rden       => bool2bit(digon(2*i+1)),
               wrclock    => POD_qclk,
               rdclock    => rclk,
               rd_aclr    => DPRAMCtrlPODReg(2),
               q          => dpram_out_pod(2*i+1)
    );

    pod_ram2 : altera_ram2p PORT MAP (
               data       => POD_IN(2*i+2),
               wren       => DPRAM_POD_write_start,
               wraddress  => pod_wraddress,
               rdaddress  => pod_rdaddress,
               rden       => bool2bit(digon(2*i+2)),
               wrclock    => POD_qclk,
               rdclock    => rclk,
               rd_aclr    => DPRAMCtrlPODReg(2),
               q          => dpram_out_pod(2*i+2)
    );

    END GENERATE;

（中略）
```

(1) オシロスコープの電圧レベル・トリガ

まず，オシロスコープの電圧レベル・トリガについて調べてみましょう．

電圧レベル・トリガはまず，被測定信号レベルが設定レベルに達した場合，変化の傾向が設定レベルを中心に立ち上がりスロープなのか，立ち下がりスロープなのかを判断します．ディジタル・オシロスコープの場合，その判定は取得したデータと設定レベルとの比較を行い，そのデータが設定レベルを超えるタイミングを検出することによって行います．今回は，サンプリングされた入力データ・ストリームが一つではなく，異なる時刻の八つのデータを同時に扱う仕組みになっているため，その最初(1番目)と最後(8番目)の値を用いて，設定値との比較を行うことにします．
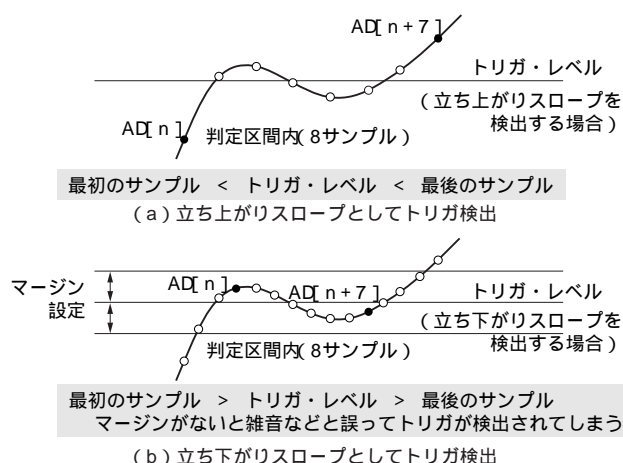
図6　オシロスコープのレベル・トリガ
ノイズの影響はシュミット・トリガの原理を用いて解決できる．

トリガは一度掛かってしまえば，初期化されるまでラッチ(latch)されます．**図6**に設計したトリガの動作を示します．この例は，被測定信号がゆっくりとした立ち上がりスロープをもつ場合のトリガ動作です．

図6(a)は立ち上がりスロープとして検出する場合です．判定区間内で，トリガ・レベルを中心に少し変化がありましたが，意図したように立ち上がりスロープがちゃんと検出されることが分かります．しかし，**図6(b)**のように，サンプリング周期が波形の変化より十分短い場合には，ノイ

リスト4　レベル・トリガのVHDL記述(level_trig.vhd)
異なる時刻に取得された八つのデータの最初(1番目)と最後(8番目)の値を用いて，設定値との比較を行う．ある程度マージン(threshold)をもたせて判定することで，トリガ動作を安定にすることができる．

```
--*************************************--
--       level_trig.vhd
--       by Minseok Kim
--*************************************--

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_UNSIGNED.all;
USE work.util_package.ALL;

ENTITY level_trig IS
  PORT (
    clock       :IN std_logic;
    reset       :IN std_logic;
    ena         :IN std_logic;

    firstD      :IN std_logic_vector(7 downto 0);
    lastD       :IN std_logic_vector(7 downto 0);
    triglev     :IN std_logic_vector(7 downto 0);
    threshold   :IN std_logic_vector(7 downto 0);

    triggered   :OUT std_logic
);
END ENTITY level_trig;
ARCHITECTURE rtl OF level_trig IS

  COMPONENT altera_comp_ge
      PORT
      (
              clock   : IN STD_LOGIC ;
              dataa   : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
              datab   : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
              AgeB    : OUT STD_LOGIC
      );
  END COMPONENT;

  COMPONENT altera_comp_lt
      PORT
      (
              clock   : IN STD_LOGIC ;
              dataa   : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
              datab   : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
              AlB     : OUT STD_LOGIC
      );
  END COMPONENT;

  SIGNAL  TRG_AlB, TRG_AgeB     :std_logic;

BEGIN

    -- トリガ・レベルより小さい? firstD < triglev - threshold
    altera_comp_lt_sig : altera_comp_lt PORT MAP (
         clock => clock,
         dataa => firstD,
         datab => triglev - threshold,  --  ノイズ・マージン
         AlB   => TRG_AlB
    );

   --  トリガ・レベルより大きい? lastD >= triglev + threshold
    altera_comp_ge_sig : altera_comp_ge PORT MAP (
         clock => clock,
         dataa => lastD,
         datab => triglev + threshold,  -- ノイズ・マージン
         AgeB  => TRG_AgeB
    );

    -- trigger after capturing at least one page
    PROCESS (TRG_AlB, TRG_AgeB, reset, ena)
      BEGIN
      IF (reset = '1') THEN
        triggered <= '0';
      ELSIF (TRG_AlB = '1' AND TRG_AgeB = '1' AND
                                          ena = '1') THEN
        triggered <= '1';
      END IF;
      END PROCESS;


END ;
```

ズなどにより誤った判定をする可能性があります．そのため，シュミット・トリガ(Schmitt trigger)の原理でもあるように，ある程度マージンをもたせて判定することで，トリガ動作を安定にすることができます．

リスト4にレベル・トリガのVHDL記述(level_trig.vhd)を示します．

(2) ロジック・アナライザのロジック・エッジ・トリガ

ロジック・アナライザにおいても，トリガの実装は必須です．ここで，簡単なロジック・トリガ機能をFPGAを用いて実装してみましょう．

オシロスコープとは異なり，ロジックの変化を検出することになります．オシロスコープではノイズなどの影響による誤判定を避ける必要がありましたが，ロジック・アナライザでは，それもグリッチ(Glitch)のようにある程度は測定の対象になります．

トリガの動作原理は入力データ・ストリームと設定されたディジタル・ロジック・パターンとの比較を行い，同じパターンが見つかった場合にトリガを掛けることになります．今回は，オシロスコープと同様に異なるタイミングの八つのデータを同時に扱う仕組みになっているため，便宜上，その最初(1番目)と最後(8番目)の値を用いて，設定値との比較を行うことにしました．

ロジック・パターンによりさまざまなトリガが実装できますが，ここでは，ベース・トリガとセカンド・トリガ(従属トリガ)の実装例を紹介します(**リスト5**)．

(3) リング状(Ring)メモリの設計

トリガの検出により，データはメモリに取り込まれます．トリガが検出された時点の前後を観測する場合，リング状のメモリ・アドレッシングがよく使われます．この場合，メモリの動的な書き込みアドレッシングが必要になります．取り込む区間の中央をトリガ時点にする場合は，トリガ時点のメモリ・アドレスは書き込み開始時のアドレスからメ

リスト5　ロジック・エッジ・トリガのVHDL記述(logic_trig.vhd)

入力データ・ストリームと設定されたディジタル・ロジック・パターンとの比較を行い，同じパターンが見つかった場合にトリガを上げる．異なるタイミングの八つのデータの最初(1番目)と最後(8番目)の値を用いて設定値との比較を行う．

```
--**********************************--
--       logic_trig.vhd
--       by Minseok Kim
--**********************************--
LIBRARY IEEE;
USE  IEEE.std_logic_1164.all;
PACKAGE type_package IS
  TYPE slv16ar8  IS ARRAY(1 to 8)  OF std_logic_vector(15
downto 0);
END type_package;

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_UNSIGNED.all;
USE work. type_package.slv16ar8;

ENTITY logic_trig IS
  PORT (
    clock      :IN std_logic;
    reset      :IN std_logic;
    ena        :IN std_logic;
    POD_D      :IN slv16ar8;

    PODTRG     :IN std_logic_vector(4 downto 0);
    PODTRGCond :IN std_logic;
    PODTRG2    :IN std_logic_vector(4 downto 0);
    PODTRGCond2:IN std_logic;

    triggered  :OUT std_logic
);
END ENTITY logic_trig;

ARCHITECTURE rtl OF logic_trig IS


  SIGNAL  ADTRGReg       : std_logic_vector(1 downto 0);
  SIGNAL  ADTRGLevelReg  : std_logic_vector(7 downto 0);

  SIGNAL  TRG1, TRG_UP,  TRG_DW   : std_logic;
  SIGNAL  TRG2, TRG_UP2, TRG_DW2  : std_logic;
  SIGNAL  TRG_AFTER_ONE_PAGE      : std_logic;
  SIGNAL  TRG_line       : std_logic_vector (1 to 2);
  SIGNAL  TRG_line2: std_logic_vector (1 to 2);

BEGIN

    -- ファスト(ベース)トリガ
    PROCESS(clock, reset, PODTRGCond, PODTRG) IS
     BEGIN
     IF (reset = '1') THEN
       TRG1 <= '0';
     ELSIF (clock'event AND clock='1') then
          --  トリガ条件が立ち上がりエッジであり,
                                そのトリガがあがった場合  OR
          --  トリガ条件が立ち下がりエッジであり,
                                そのトリガがあがった場合  OR
          --  フリーランの場合
                          (ただし，メモリは一回はフルに書いてから)
       IF (PODTRGCond = '0' AND TRG_UP='1') OR
(PODTRGCond = '1' AND TRG_DW='1') OR
          (PODTRG = "00000" AND TRG_AFTER_ONE_PAGE ='1')
                                                        THEN
         TRG1 <= '1';
       END IF;
     END IF;
     END PROCESS;

    -- セカンド・トリガ
    PROCESS(clock, reset, PODTRGCond2, PODTRG2) IS
     BEGIN
     IF (reset = '1') THEN
       TRG2 <= '0';
     ELSIF (clock'event AND clock='1') then
       IF (PODTRGCond2 = '0' AND TRG_UP2='1') OR
                   (PODTRGCond2 = '1' AND TRG_DW2='1') OR
          (PODTRG2 = "00000" AND TRG_AFTER_ONE_PAGE
                                              ='1') THEN
         TRG2 <= '1';
       END IF;
```

リスト5　ロジック・エッジ・トリガのVHDL記述(logic_trig.vhd)（つづき）

```
      END IF;
      END PROCESS;

    -- 総合的なトリガ
    PROCESS(clock, reset) IS
      BEGIN
      IF (reset = '1') THEN
        triggered <= '0';
      ELSIF (clock'event AND clock='1') then
        IF TRG1 = '1' THEN
          triggered <= TRG2;
        END IF;
      END IF;
      END PROCESS;


--     --トリガ・ライン・セレクタ：ここでは1サンプル目と8サンプル
目の状況をみて判断
--
--
__|~~|__|~~|__|~~|__|~~|__|~~|__|~~|__|~~|__|~~|__|~~|__|
~~|__|~~|__|~~|__
--                    |< 1 >|< 2 >|< 3 >|< 4 >|< 5 >|< 6
>|< 7 >|< 8 >|

    PROCESS(PODTRG, POD_D, clock)
      BEGIN
        IF (clock'event AND clock='1') THEN
        CASE PODTRG IS
               WHEN "00001" => TRG_line <= POD_D(1)( 0) &
                                            POD_D(8)( 0);
               WHEN "00010" => TRG_line <= POD_D(1)( 1) &
                                            POD_D(8)( 1);
               WHEN "00011" => TRG_line <= POD_D(1)( 2) &
                                            POD_D(8)( 2);
               WHEN "00100" => TRG_line <= POD_D(1)( 3) &
                                            POD_D(8)( 3);
               WHEN "00101" => TRG_line <= POD_D(1)( 4) &
                                            POD_D(8)( 4);
               WHEN "00110" => TRG_line <= POD_D(1)( 5) &
                                            POD_D(8)( 5);
               WHEN "00111" => TRG_line <= POD_D(1)( 6) &
                                            POD_D(8)( 6);
               WHEN "01000" => TRG_line <= POD_D(1)( 7) &
                                            POD_D(8)( 7);
               WHEN "01001" => TRG_line <= POD_D(1)( 8) &
                                            POD_D(8)( 8);
               WHEN "01010" => TRG_line <= POD_D(1)( 9) &
                                            POD_D(8)( 9);
               WHEN "01011" => TRG_line <= POD_D(1)(10) &
                                            POD_D(8)(10);
               WHEN "01100" => TRG_line <= POD_D(1)(11) &
                                            POD_D(8)(11);
               WHEN "01101" => TRG_line <= POD_D(1)(12) &
                                            POD_D(8)(12);
               WHEN "01110" => TRG_line <= POD_D(1)(13) &
                                            POD_D(8)(13);
               WHEN "01111" => TRG_line <= POD_D(1)(14) &
                                            POD_D(8)(14);
               WHEN "10000" => TRG_line <= POD_D(1)(15) &
                                            POD_D(8)(15);
               WHEN OTHERS  => TRG_line <= "11";
        END CASE;
      END IF;
    END PROCESS;

    PROCESS(PODTRG2, POD_D, clock)
      BEGIN
        IF (clock'event AND clock='1') THEN
        CASE PODTRG2 IS
               WHEN "00001" => TRG_line2 <= POD_D(1)( 0)
                                          & POD_D(8)( 0);
               WHEN "00010" => TRG_line2 <= POD_D(1)( 1)
                                          & POD_D(8)( 1);
               WHEN "00011" => TRG_line2 <= POD_D(1)( 2)
                                          & POD_D(8)( 2);
               WHEN "00100" => TRG_line2 <= POD_D(1)( 3)
                                          & POD_D(8)( 3);
               WHEN "00101" => TRG_line2 <= POD_D(1)( 4)
                                          & POD_D(8)( 4);
               WHEN "00110" => TRG_line2 <= POD_D(1)( 5)
                                          & POD_D(8)( 5);
               WHEN "00111" => TRG_line2 <= POD_D(1)( 6)
                                          & POD_D(8)( 6);
               WHEN "01000" => TRG_line2 <= POD_D(1)( 7)
                                          & POD_D(8)( 7);
               WHEN "01001" => TRG_line2 <= POD_D(1)( 8)
                                          & POD_D(8)( 8);
               WHEN "01010" => TRG_line2 <= POD_D(1)( 9)
                                          & POD_D(8)( 9);
               WHEN "01011" => TRG_line2 <= POD_D(1)(10)
                                          & POD_D(8)(10);
               WHEN "01100" => TRG_line2 <= POD_D(1)(11)
                                          & POD_D(8)(11);
               WHEN "01101" => TRG_line2 <= POD_D(1)(12)
                                          & POD_D(8)(12);
               WHEN "01110" => TRG_line2 <= POD_D(1)(13)
                                          & POD_D(8)(13);
               WHEN "01111" => TRG_line2 <= POD_D(1)(14)
                                          & POD_D(8)(14);
               WHEN "10000" => TRG_line2 <= POD_D(1)(15)
                                          & POD_D(8)(15);
               WHEN OTHERS  => TRG_line2 <= "11";
        END CASE;
      END IF;
    END PROCESS;

    -- For freerun(トリガなしのフリーラン)
    PROCESS (TRG_line, reset, ena)
      BEGIN
      IF(reset = '1') THEN
        TRG_AFTER_ONE_PAGE <= '0';
      ELSIF (TRG_line = "11" AND ena = '1' ) THEN
        TRG_AFTER_ONE_PAGE <= '1';
      END IF;
      END PROCESS;

    -- 立ち上がりエッジ・トリガ(ベース・トリガ)
    PROCESS (TRG_line, reset, ena)
      BEGIN
      IF (reset = '1') THEN
        TRG_UP <= '0';
      ELSIF (TRG_line(1) = '0' AND TRG_line(2) = '1' AND
                                          ena = '1') THEN
        TRG_UP <= '1';
      END IF;
      END PROCESS;

    -- 立ち下がりエッジ・トリガ(ベース・トリガ)
    PROCESS (TRG_line, reset, ena)
      BEGIN
      IF(reset = '1') THEN
        TRG_DW <= '0';
      ELSIF (TRG_line(1) = '1' AND TRG_line(2) = '0' AND
                                          ena = '1') THEN
        TRG_DW <= '1';
      END IF;
      END PROCESS;

    -- 立ち上がりエッジ・トリガ(セカンド・トリガ)
    PROCESS (TRG_line2, reset)
      BEGIN
      IF (reset = '1') THEN
        TRG_UP2 <= '0';
      ELSIF (TRG_line2(1) = '0' AND TRG_line2(2) = '1')
                                                    THEN
        TRG_UP2 <= '1';
      END IF;
      END PROCESS;

     -- 立ち下がりエッジ・トリガ(セカンド・トリガ)
    PROCESS (TRG_line2, reset)
      BEGIN
      IF(reset = '1') THEN
        TRG_DW2 <= '0';
      ELSIF (TRG_line2(1) = '1' AND TRG_line2(2) = '0')
                                                    THEN
        TRG_DW2 <= '1';
      END IF;
      END PROCESS;

END ;
```

モリ・サイズの半分だけオフセットされたアドレスになります．

今回の設計は，512ワードのデュアル・ポートRAMを基本コンポーネントにしているので，トリガ時点は255番地として動的に扱うことになります．

図7にトリガ動作によるメモリ書き込みの流れと，リング状のメモリ・アドレッシングの仕組みを示します．まず，測定データを取得する前にメモリ初期化を行い，キャプチャ開始信号を発行します．次に，メモリ制御用のFPGAの内部レジスタの書き込みイネーブルがセットされ，メモリは書き込みを開始します．そして，入力データからトリガが検出されたら，その時点から書き込みクロック周期をメモリ・サイズの半分(255)までカウント・アップしていきます．カウンタが255になれば，書き込みイネーブルをリセットさせ，書き込み動作を終了します．実際にメモリ回路は八つ並列で構成されますが(総4096ワード＝512×8)，すべてのメモリが同じ動作をすればよいことになります．

今回の設計は「オシロスコープ機能付きロジック・アナライザ」なので，ロジック・アナライザでの観測と同時にオシロスコープでアナログ的に観測する機能を実装してみます．そのため，A-Dコンバータの入力段にアナログ・マルチプレクサを搭載し，ロジック・アナライザのPODごとに8ビットのうち下3ビットのラインとオシロスコープのプローブをソフトウェアで選択できるようにしています．また，ロジック・アナライザのトリガに，オシロスコープのトリガを同期させるように実装すれば，ディジタルとアナログの同時測定が実現できます．

3. マイコンとのインターフェース設計

今回のシステムは，マイコンとFPGAの間はSPI(serial peripheral interface)方式のシリアル通信を行います．使用するマイコンは，2チャネルのSPIコントローラを内蔵していますが，そのうち1チャネルのみをSPI通信用に，残りはGPIO(general purposed I/O)として使います．CPUとFPGA間のデータのやり取りには，4本のSPIシリアル・インターフェースを用います．

● SPIスレーブ・コントローラ

SPIは米国Motorola社が提案した3線(クロック，マスタ出力データ，マスタ入力データ)シリアル通信方式であり，現在は多くのLSIに内蔵されています．

SPIコントローラは，**図8**のようにマスタ(master)とスレーブ(slave)として構成します．マスタはチップ・セレクト信号(nCS)を発行し，スレーブの選択を行います．そして，シリアル・クロック(SCLK)に同期してマスタ・コントローラのシフトレジスタのデータが，MSB (most significant bit)からMOSI(master-out slave-in)よりスレーブに送られ，それと同時にMISO(master-in slave-out)よりスレーブのデータを受信するといった非常に単純
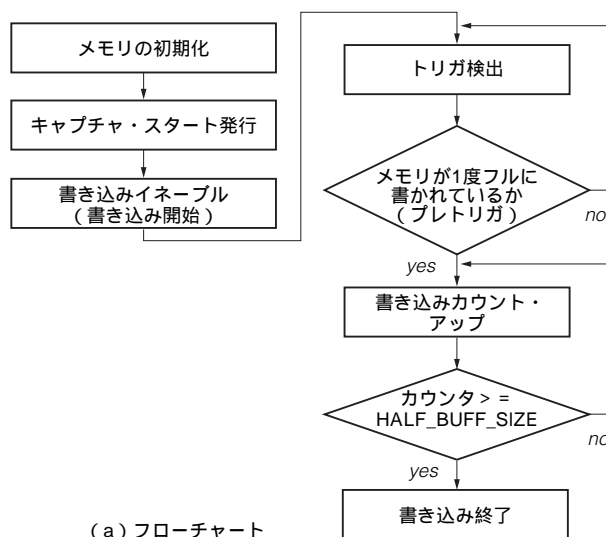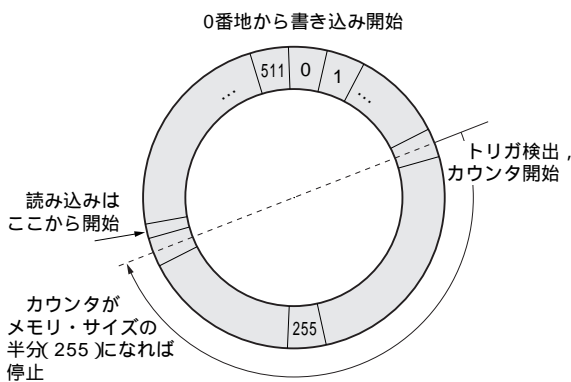
図7 トリガ動作によるメモリ書き込みの流れとリング状のメモリ・アドレッシング
トリガ時点を取り込む区間の中央にし，トリガ時点は255番地として動的に扱う．
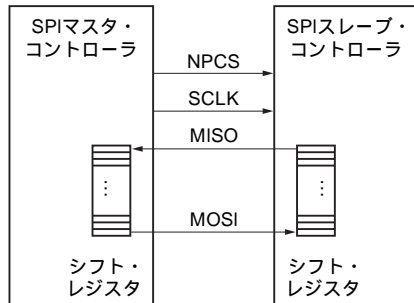
図8　SPIシリアル通信方式（スレーブが1個のみの場合）

マスタはチップ・セレクト信号（nCS）を発行しスレーブの選択を行い，シリアル・クロック（SCLK）に同期してマスタ・コントローラのシフトレジスタのデータが，MSBからMOSIよりスレーブに送られ，それと同時にMISOよりスレーブのデータを受信する．

な方式です．

本システムでは，CPUがマスタ・モード（master mode）としてnCSやSCLKを発行し，FPGAはスレーブ（slave）になり，CPUとのデータのやり取り（FPGA内部レジスタやメモリの書き込み・読み出し）を行います．FPGAのSPIスレーブ・コントローラは，16ビット・シフトレジスタを制御するステート・マシン（80MHz動作）を用いて実装しました（**リスト6**）．

● SPIコマンド形式

今回の設計では，16ビット・データのやり取りをベース

リスト6　SPIスレーブ（Slave）コントローラのVHDL記述（SPIctrl.vhd）

FPGAのSPIのスレーブ・コントローラには16ビット・シフトレジスタを制御するステート・マシン（80MHz動作）が用いられる．

```
--*************************************--
--         SPIctrl.vhd
--       by Minseok Kim
--*************************************--

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_UNSIGNED.all;
USE work.util_package.ALL;

ENTITY SPIctrl IS
  PORT (
    clock              :IN std_logic;
    SPI_SPCK           :IN std_logic;
    SPI_MOSI           :IN std_logic;
    SPI_NPCS           :IN std_logic;
    SPI_MISO           :OUT std_logic;
    rdreq_noti         :OUT std_logic;
    rdreq              :OUT std_logic;
    wrreq              :OUT std_logic;
    is_SPI_state_idle  :OUT std_logic;
    SPI_ADDR           :OUT std_logic_vector(7 downto 0);
    SPI_D_to_Mst       :IN std_logic_vector(15 downto 0);
    SPI_D_from_Mst     :OUT std_logic_vector(15 downto 0)
);
END ENTITY SPIctrl;

ARCHITECTURE rtl OF SPIctrl IS

  CONSTANT  SPI_BITS     : Integer := 16;
  CONSTANT  SPI_BITS_HALF: Integer := 8;
  CONSTANT  RISING_EDGE  : std_logic_vector := "01";

  SIGNAL  SHIFTREG  : std_logic_vector(SPI_BITS-1 downto
                                0);  -- Transmit register
  SIGNAL  bcnt      : std_logic_vector(4 downto 0);
  TYPE SPIstate IS
      (STATE_SPI_IDLE,
       STATE_SPI_CMD_SHIFT,
       STATE_SPI_CMD_LOAD_FROM_SR,
       STATE_SPI_DATA_LOAD_RDREQ,
       STATE_SPI_CMD_LOAD_FROM_SR_REG,
       STATE_SPI_DATA_LOAD_TO_SR,
       STATE_SPI_DATA_SHIFT,
       STATE_SPI_DATA_WRITE_REG,
       STATE_SPI_WAIT_CS_UP
       );
  SIGNAL spi_state : SPIstate;

  SIGNAL  rwflag       : std_logic;
  SIGNAL  idle_state_on, sr_load_on         : std_logic;

  SIGNAL  SPI_ADDR_t              :
                   std_logic_vector(SPI_BITS_HALF-1 downto 0);
  SIGNAL  burst_mode      : std_logic;
  SIGNAL  SPI_SPCK_BUF    : std_logic_vector(1 downto 0);

BEGIN

   --  シリアル・クロックをレジスタする
   PROCESS (clock) IS
     BEGIN
       IF clock'event and clock = '1' THEN
         SPI_SPCK_BUF(1) <= SPI_SPCK_BUF(0);
         SPI_SPCK_BUF(0) <= SPI_SPCK;
       END IF;
     END PROCESS;

  -- SPI データ・トランスファ  ステート・マシン
   PROCESS (clock, SPI_NPCS, bcnt) IS
   BEGIN
     IF clock'event and clock = '1' THEN
       CASE spi_state IS
         WHEN STATE_SPI_IDLE =>
            IF SPI_NPCS = '0' THEN
--  スレーブとして選択されたら
              idle_state_on <= '0';
              spi_state <= STATE_SPI_CMD_SHIFT;
            ELSE
              idle_state_on <= '1';
              spi_state    <= STATE_SPI_IDLE;
            END IF;
            wrreq         <= '0';
            rdreq_noti    <= '0';
            rwflag        <= '0';
            sr_load_on    <= '0';
            rdreq         <= '0';
            burst_mode    <= '0';
            SPI_ADDR_t    <= (OTHERS=>'0');

         WHEN STATE_SPI_CMD_SHIFT =>
            IF (bcnt = "00000" ) THEN
--  シフトレジスタがフルになるまでシフト
              spi_state <= STATE_SPI_CMD_LOAD_FROM_SR;
            ELSE
              spi_state <= STATE_SPI_CMD_SHIFT;
            END IF;

         WHEN STATE_SPI_CMD_LOAD_FROM_SR =>
            SPI_ADDR_t <= SHIFTREG(SPI_BITS_HALF-1
                                   downto 0);   --  アドレス
            rwflag     <= SHIFTREG(SPI_BITS-1);
                                   --  リード・ライト・フラグ
            burst_mode <= SHIFTREG(SPI_BITS-2);
                                 --  バースト・モード・フラク
            spi_state <= STATE_SPI_DATA_LOAD_RDREQ;
```

とし，**図9**のようにパケット・フォーマットを定義しています．実際は，OSとの整合性を取るために，CPUからは8ビット・ワードの2回転送を行う（マイコンのアプリケーションにて実装）ことにしています．

コマンドの構成上，MSBはリード・ライト・フラグ，15ビット目はA-Dコンバータのデータと POD データを1回のコマンドでリードするためのバースト・フラグであり，下位8ビットからはレジスタのアドレスとなります．

表1に今回のシステムで用いたFPGA内部レジスタの構成例と簡単な仕様を示します．

R/Wフラグ（1ビット）	バースト・リード・フラグ（1ビット）	予備（6ビット）	レジスタ・アドレス（8ビット）

図9　SPIコマンド・フォーマット

リスト6　SPIスレーブ（Slave）コントローラのVHDL記述（SPIctrl.vhd）（つづき）

```
        WHEN STATE_SPI_DATA_LOAD_RDREQ =>
           rdreq        <= NOT rwflag;
                   --  リードなら，リード・リクエストを立てる
           spi_state <= STATE_SPI_CMD_LOAD_FROM_SR_REG;

        WHEN STATE_SPI_CMD_LOAD_FROM_SR_REG =>
           SPI_ADDR_t <= SPI_ADDR_t;
           rwflag      <= rwflag;
           rdreq       <= '0';
           sr_load_on <= '1';
                          --  シフトレジスタ・ロード・フラク
           spi_state <= STATE_SPI_DATA_LOAD_TO_SR;

        WHEN STATE_SPI_DATA_LOAD_TO_SR =>
           IF rwflag = '0' THEN   --  リードなら
             rdreq_noti <= '1';
                 --  リード・リクエスト・ノティファイアを立てる
           END IF;
        sr_load_on <= '0';
           spi_state <= STATE_SPI_DATA_SHIFT;

        WHEN STATE_SPI_DATA_SHIFT =>
           IF (bcnt = "00000" ) THEN
                    --  シフトレジスタがフルになるまでシフト
             IF rwflag = '0' THEN      -- リードなら
               rdreq_noti  <= '0';
               spi_state <= STATE_SPI_WAIT_CS_UP;
             ELSE                  -- ライトなら
               spi_state <= STATE_SPI_DATA_WRITE_REG;
             END IF;
           ELSE
             spi_state <= STATE_SPI_DATA_SHIFT;
           END IF;

        WHEN STATE_SPI_DATA_WRITE_REG =>
           wrreq  <= '1';     --  ライト・リクエストを立てる
           spi_state <= STATE_SPI_WAIT_CS_UP;

        WHEN STATE_SPI_WAIT_CS_UP =>
           wrreq  <= '0';
           IF SPI_NPCS = '1' THEN
                   --  スレーブ選択が解除されるまで待てて終了
             IF burst_mode = '1' THEN
                              --  バースト転送モードなら
               IF SPI_ADDR_t = X"1F" THEN
                 --  アドレスがデータフレームの最後の場合に終了
                 spi_state <= STATE_SPI_IDLE;
               ELSE
                 SPI_ADDR_t  <= SPI_ADDR_t + '1';
                             --  アドレス・インクリメント
                 spi_state <= STATE_SPI_DATA_LOAD_RDREQ;
               END IF;
             ELSE
               spi_state <= STATE_SPI_IDLE;
             END IF;
           ELSE
             spi_state <= STATE_SPI_WAIT_CS_UP;
           END IF;

        WHEN OTHERS =>
           spi_state <= STATE_SPI_IDLE;
             END CASE;
           END IF;
         END PROCESS;

--  シフトレジスタ記述
  PROCESS (SPI_SPCK_BUF, clock, idle_state_on,
sr_load_on, rwflag) IS
    BEGIN
      IF clock'event AND clock = '1' THEN
        IF idle_state_on = '1' THEN
          SHIFTREG  <= (OTHERS=>'0'); -- load transfer
                                               register
          bcnt    <= "10000"; -- set transfer counter

        ELSIF sr_load_on = '1' AND rwflag = '0' THEN
                                             -- リードなら
          SHIFTREG  <= SPI_D_to_Mst;
                   -- マスタに送るデータをシフトレジスタにロード
          bcnt    <= "10000"; -- set transfer counter(16)

        ELSIF sr_load_on = '1' AND rwflag = '1' THEN
                                            --  ライトなら
          SHIFTREG  <= (OTHERS=>'0');
             -- マスタからデータを受信するために，空のデータを送る
          bcnt    <= "10000"; -- set transfer counter(16)

        ELSIF SPI_SPCK_BUF = RISING_EDGE THEN
          SHIFTREG  <= SHIFTREG(SPI_BITS-2 downto 0) &
                  SPI_MOSI;  --  シリアル・クロックの立ち上がり
                                --  エッジで同期してシフト・イン
          bcnt      <= bcnt - '1';

        ELSE
          SHIFTREG  <= SHIFTREG;
          bcnt      <= bcnt;

        END IF;
      END IF;
    END PROCESS;


    PROCESS (SPI_SPCK) IS
      BEGIN
        IF SPI_SPCK'event and SPI_SPCK = '0' THEN
        SPI_MISO <= SHIFTREG(SPI_BITS-1);
--  シリアル・クロックの立ち下がりエッジに
--  同期してシフト・イン
        END IF;
      END PROCESS;

    SPI_D_from_Mst <= SHIFTREG;
    SPI_ADDR        <= SPI_ADDR_t;
    is_SPI_state_idle <= idle_state_on;

END ;
```

表1　FPGA内部レジスタの構成例

アドレス	レジスタ名	説　明	アクセス	使用例(from lsb)
0x00	TestReg	テスト・レジスタ	W/R	16ビット
0x01	SampleFreqReg	ADCサンプル・レート・レジスタ	W	4ビット
0x02	PLL_PS_Reg	PLL PSレジスタ	W	2ビット PLL_ZC_Reg(1) PLL_PS_Reg(0)
0x03	DPRAMCtrlPODReg	RAM制御レジスタ(POD)	W/R	3ビット(reset, wena, rena)
0x04	AD1SelReg	マルチ・プレクサ・セレクタAD1	W	2ビット(0d～3d)
0x05	AD2SelReg	マルチ・プレクサ・セレクタAD2	W	2ビット(0d～3d)
0x06	SerCtrlSelReg	シリアル・チャネル・セレクタ(PLL/ADC)	W	2ビット(0d～2d)
0x07	SampleFreqPODReg	PODサンプル・レート・レジスタ	W	4ビット
0x08	DPRAMCtrlADReg	RAM制御レジスタ(AD)	W/R	3ビット(reset, wena, rena)
0x09	CaptureStart	キャプチャ・スタート・レジスタ	W	1ビット
0x10～0x1E(even)	AD_D_Reg(#1~#8)	現在の読み込みアドレスのADデータ	R	16ビット
0x11～0x1F(odd)	POD_D_Reg(#1~#8)	現在の読み込みアドレスのPODデータ	R	16ビット
0x30	POD_TRIG_Reg	PODトリガ・レジスタ	W	16ビット PODTRGReg[4..0] PODTRGCondReg[5] PODTRGReg2[12..8] PODTRGCondReg2[13]
0x31	AD_TRIG_Reg	ADトリガ・レジスタ	W	4ビット ADTRGReg[1..0] AD_FreeRUN[2] AD_PODSync[3]
0x32	AD_TRIG_Level_Reg	ADトリガ・レベル・レジスタ	W	8ビット
0x33	Start_RD_ADDR_Reg	読み込みスタート・アドレス・レジスタ	W	16ビット(0d～4095d)
0x34～0xff	―	予約済み	―	―

W：書き込み，R：読み出し
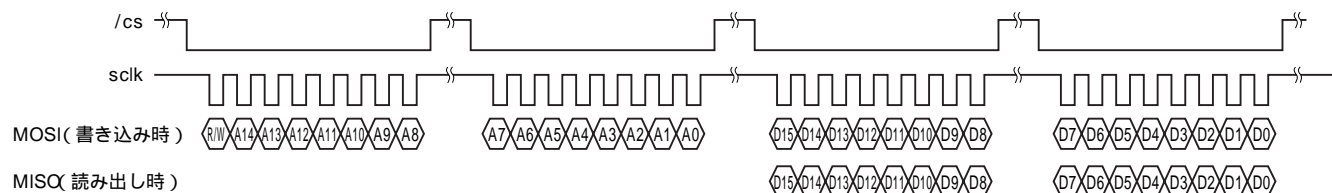
図10　SPIコマンド・モード(レジスタ書き込み/読み出し)
2バイトのコマンド(アドレス)と2バイトのデータの4バイト構成．

●データ転送モード

データ転送モードは，FPGAの内部レジスタの読み書きを行うためのコマンド・モードと，A-D変換データやPODデータのバースト転送を行うためのバースト・モードを定義しています．

コマンド・モードは，**図10**のように2バイトのコマンド(アドレス)と2バイトのデータで4バイト構成になります．

バースト・モードはバースト・モード・フラグを立てて，AD_D_Reg(チャネル1)の1番目のアドレスにアクセスします．すると，FPGAは自動的にアドレスをインクリメントし，メモリのデータを連続して転送します．

バースト・フレームは，以下のように32バイト構成とし，その転送タイミング・チャートは**図11**のようになります．

> バースト・フレーム：{AD2・AD1・POD2・POD1}の4バイト×八つのメモリ・アクセス

4．FPGAの実装結果

今回設計したLVDS差動インターフェースのSerDesとDDRデータ・キャプチャ回路を，Quartus IIでコンパイル
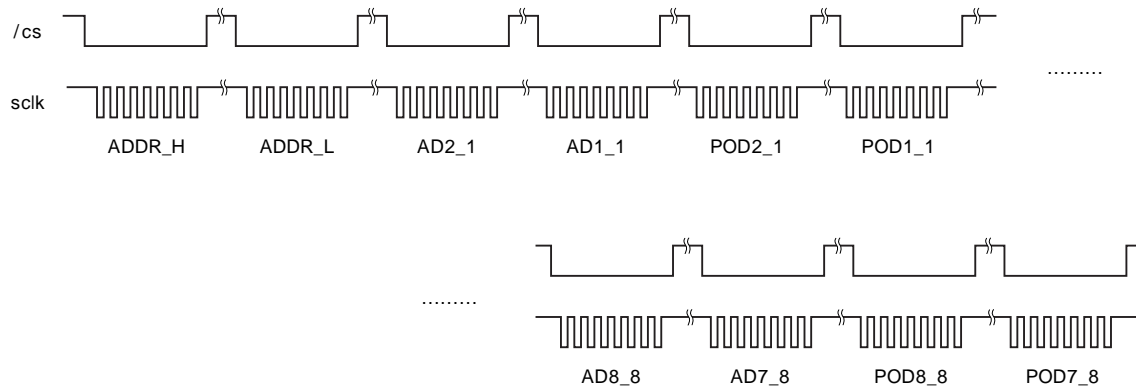
図11 SPIバースト・モード(ADとPODデータの4チャネル分を連続して読み出し)
バースト・モードでAD_D_Reg(チャネル1)の1番目のデータを読み出す際,FPGAは自動的にアドレスのインクリメントを行い,メモリのデータを連続に転送する.バースト・フレームは32バイトで構成される.
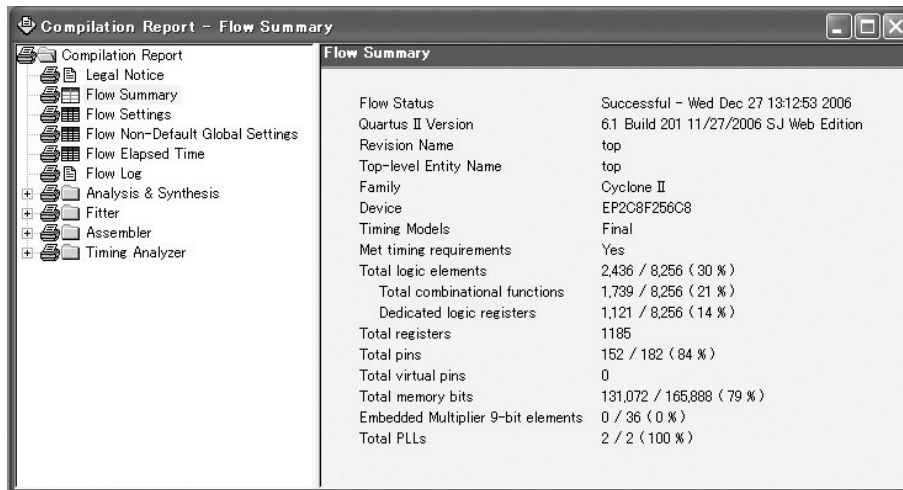
図12 FPGAの実装結果(Quartus II Web Edition バージョン6.1)
Quartus IIのコンパイル・レポート.論理ブロックとメモリの使用量はそれぞれ30%と80%程度.PLLはLVDS SerDes用として1個,内部クロック管理用として1個使用.

したFPGAの実装結果を,**図12**に示します.論理ブロックとメモリの使用量はそれぞれ30%,80%程度になっています.

PLLはLVDS SerDes用として1個,内部クロック・マネジメント用として1個使用します.今回紹介した「オシロスコープ機能付きロジック・アナライザ」の設計例から,低コストFPGAで提供される機能を十分に活用すれば,簡単な測定器まで実現できることが分かります.

今回は掲載できませんでしたが,リアルタイムOSを搭載し,GUIを開発するところまでを,機会があれば紹介したいと思います.

Minseok Kim
(株)ブレインズ